Tests, Backtracking, and Recursion

Artificial Intelligence Programming in Prolog
Lecture 3
30/09/04
Re-cap

- A Prolog program consists of predicate definitions.
- A predicate denotes a property or relationship between objects.
- Definitions consist of clauses.
- A clause has a head and a body (Rule) or just a head (Fact).
- A head consists of a predicate name and arguments.
- A clause body consists of a conjunction of terms.
- Terms can be constants, variables, or compound terms.
- We can set our program goals by typing a command that unifies with a clause head.
- A goal unifies with clause heads in order (top down).
- Unification leads to the instantiation of variables to values.
- If any variables in the initial goal become instantiated this is reported back to the user.
Correction: Re-trying Goals

- When a question is asked with a variable as an argument (e.g. `greet(Anybody).`) we can ask the Prolog interpreter for multiple answers using:

```prolog
| ?- greet(Anybody).
  How are you doin, pal?
  Anybody = hamish? ;
  Awfully nice to see you!
  Anybody = amelia? ;
  no
```

- `;` fails the last clause used and searches down the program for another that matches.
- It then performs all the tasks contained within the body of the new clause and returns the new value of the variable.
Tests

• When we ask Prolog a question we are asking for the interpreter to prove that the statement is true.

?− 5 < 7, integer(bob).

yes = the statement can be proven.

no = the proof failed because either
   – the statement does not hold, or
   – the program is broken.

Error = there is a problem with the question or program.

*nothing* = the program is in an infinite loop.

• We can ask about:
  – Properties of the database: mother(jane, alan).
  – Built-in properties of individual objects: integer(bob).
  – Absolute relationships between objects:
    • Unification: =/2
    • Arithmetic relationships: < , > , =< , >= , =:= , + , - , * , /
Arithmetic Operators

- Operators for arithmetic and value comparisons are **built-in** to Prolog
  - = always accessible / don’t need to be written
- Comparisons: <, >, <=, >=, =:= (equals), =\= (not equals)
  - Infix operators: go between two terms.
  - <=/2 is used
    - 5 <= 7. (infix)
    - <=(5, 7). (prefix) ← all infix operators can also be prefixed
- Equality is different from unification
  - /=2 checks if two terms unify
  - =:=/2 compares the arithmetic value of two expressions
    - ?- X=:=Y.
    - ?- X=:=Y.
    - ?-X=4,Y=3, X+2 =:= Y+3.
    - yes Instantiation error X=4, Y=3? yes
Arithmetic Operators (2)

- Arithmetic Operators: +, -, *, /
  = Infix operators but can also be used as prefix.
  - Need to use `is/2` to access result of the arithmetic expression otherwise it is treated as a term:

```
?- X = 5+4.  X = 5+4 ?  yes
?- X is 5+4.  X = 9 ?  yes
(Can X unify with 5+4?)  (What is the result of 5+4?)
```

- Mathematical precedence is preserved: /, *, before +,-
- Can make compound sums using round brackets
  - Impose new precedence
  - Inner-most brackets first

```
?- X is (5+4)*2.  X = 18 ?  yes
```
Tests within clauses

- These operators can be used within the body of a clause:
  - To manipulate values,
    ```prolog
    sum(X,Y,Sum):-
    Sum is X+Y.
    ```
  - To distinguish between clauses of a predicate definition
    ```prolog
    bigger(N,M):-
    N < M, write('The bigger number is '), write(M).
    bigger(N,M):-
    N > M, write('The bigger number is '), write(N).
    bigger(N,M):-
    N =:= M, write('Numbers are the same').
    ```
Backtracking

?- bigger(5,4).

bigger(N,M):-
    N < M,
    write('The bigger number is '), write(M).

bigger(N,M):-
    N > M,
    write('The bigger number is '), write(N).

bigger(N,M):-
    N =:= M,
    write('Numbers are the same').
Backtracking

?- bigger(5,4).

5 < 4, \(\Leftarrow\) fails

write(‘The bigger number is ‘), write(M).

bigger(N,M):-
    N > M,
    write(‘The bigger number is ‘), write(N).

bigger(N,M):-
    N =:= M,
    write(‘Numbers are the same’).
Backtracking

?- bigger(5,4).

bigger(N,M):-
    N < M,
    write('The bigger number is '), write(M).

bigger(5,4):-
    5 > 4,
    write('The bigger number is '), write(N).

bigger(N,M):-
    N =:= M,
    write('Numbers are the same').
|?- bigger(5,4).

bigger(N,M):-
    N < M,
    write('The bigger number is '), write(M).

bigger(5,4):-
    5 > 4, ← succeeds, go on with body.
    write('The bigger number is '), write(5).

The bigger number is 5
yes
|?-
Backtracking

?- bigger(5,5). ← If our query only matches the final clause

bigger(N,M):-
    N < M,
    write('The bigger number is '), write(M).

bigger(N,M):-
    N > M,
    write('The bigger number is '), write(N).

bigger(5,5):-
    5 =:= 5, ← Is already known as the first two clauses failed.
    write('Numbers are the same').
Backtracking

?- bigger(5,5).
  \[\text{If our query only matches the final clause}\]

\[
\text{bigger}(N,M):- \\
\quad \text{N} < \text{M}, \\
\quad \text{write(‘The bigger number is ‘)}, \text{write(M)}.
\]

\[
\text{bigger}(N,M):- \\
\quad \text{N} > \text{M}, \\
\quad \text{write(‘The bigger number is ‘)}, \text{write(N)}.
\]

\[
\text{bigger}(5,5):- \\
\text{\hspace{1cm} \hspace{1cm} Satisfies the same conditions.} \\
\quad \text{write(‘Numbers are the same‘)}.
\]

Numbers are the same

\text{yes}

\boxed{\text{Clauses should be ordered according to specificity}} \\
\text{Most specific at top} \quad \text{Universally applicable at bottom}
Reporting Answers

\(?- \text{bigger}(5,4)\).
\text{The bigger number is 5}
\text{yes}

\(?- \text{bigger}(6,4), \text{bigger}(\text{Answer},5)\).
\text{Instantiation error!}

- This is fine for checking what the code is doing but not for using the proof.

- To report back answers we need to
  - put an uninstantiated variable in the query,
  - instantiate the answer to that variable when the query succeeds,
  - pass the variable all the way back to the query.
Passing Back Answers

- To report back answers we need to
  1. put an uninstantiated variable in the query,
  2. instantiate the answer to that variable when the query succeeds,
  3. pass the variable all the way back to the query.

\[
\begin{align*}
\text{bigger}(X,Y,\text{Answer}) :&= X > Y, \quad \text{Answer} = X. \\
\text{bigger}(X,Y,\text{Answer}) :&= X \leq Y, \quad \text{Answer} = Y.
\end{align*}
\]
Head Unification

- To report back answers we need to
  1. put an **uninstantiated variable** in the query,

|   ?- bigger(6,4,Answer),bigger(Answer,5,New_answer). |

Or, do steps 2 and 3 in one step by naming the variable in the head of the clause the same as the correct answer.

= **head unification**
Satisfying Subgoals

- Most rules contain calls to other predicates in their body. These are known as **Subgoals**.
- These subgoals can match:
  - facts,
  - other rules, or
  - the same rule = a recursive call

1) drinks(alan,beer).
2) likes(alan,coffee).
3) likes(heather,coffee).

4) likes(Person,Drink):-
   drinks(Person,Drink).  ← a different subgoal
5) likes(Person,Somebody):-
   likes(Person,Drink),  ← recursive subgoals
   likes(Somebody,Drink).  ←
Representing Proof using Trees

- To help us understand Prolog’s proof strategy we can represent its behaviour using AND/OR trees.

1. Query is the top-most point (node) of the tree.
2. Tree grows downwards (looks more like roots!).
3. Each branch denotes a subgoal.
   1. The branch is labelled with the number of the matching clause and
   2. any variables instantiated when matching the clause head.
4. Each branch ends with either:
   1. A successful match ✓,
   2. A failed match ✗, or
   3. Another subgoal.

<table>
<thead>
<tr>
<th>?- likes(alan,X).</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 X/coffee</td>
</tr>
<tr>
<td>X = coffee</td>
</tr>
</tbody>
</table>

1st solution = “Alan likes coffee.”
Representing Proof using Trees (2)

- Using the tree we can see what happens when we ask for another match (

|?- likes(alan,X).

\[ X = \text{coffee} \]

\[ X = \text{beer} \]

1st match is failed and forgotten

2nd solution

= “Alan likes beer because Alan drinks beer.”
Recursion using Trees

- When a predicate calls itself within its body we say the clause is **recursing**

```
?- likes(alan,X).
```

- `likes(alan,Drink)`
- `drinks(alan,X).`
  - `X/coffee`
  - `X = coffee`

```
| ?- likes(alan,X).
|   2
|   X/coffee
|   4
|   X = coffee
```

```
| ?- likes(alan,Drink).
|   5
|   likes(alan,Drink)
|   likes(Somebody,Drink)
```

```
| ?- likes(alan,Drink).
|   2
|   X/coffee
```

```
| ?- likes(alan,Drink).
|   1
|   X/beer

| ?- likes(alan,Drink).
|   1
|   X = beer
```

```
| ?- likes(alan,Drink).
|   X = coffee
```

Conjoined subgoals
Recursion using Trees (2)

- When a predicate calls itself within its body we say the clause is **recursing**

```
|?- likes(alan,X).

X/coffee  2
X = coffee

|?- drinks(alan,X).

likes(alan,coffee)

|?- likes(Somebody,coffee).

X/coffee  2
X = coffee

|?- likes(alan,coffee).

X/coffee  2
X = coffee
```

3rd solution = “Alan likes Alan because Alan likes coffee.”
Recursion using Trees (3)

- When a predicate calls itself within its body we say the clause is **recursing**

```
?- likes(alan,X).
```

4th solution =
“Alan likes Heather because Heather likes coffee.”
Infinite Recursive Loop

- If a recursive clause is called with an incorrect goal it will loop as it can neither prove it nor disprove it.
The central ideas of Prolog

• SUCCESS/FAILURE
  – any computation can “succeed” or “fail”, and this is used as a ‘test’ mechanism.

• MATCHING
  – any two data items can be compared for similarity, and values can be bound to variables in order to allow a match to succeed.

• SEARCHING
  – the whole activity of the Prolog system is to search through various options to find a combination that succeeds.
    • Main search tools are backtracking and recursion

• BACKTRACKING
  – when the system fails during its search, it returns to previous choices to see if making a different choice would allow success.